Type Systems

CMSC 631 – Program Analysis and Understanding
Spring 2009

The Need for a Type System

• Consider the (untyped) lambda calculus
  ▪ false = \( \lambda x.\lambda y.x \)
  ▪ 0 (Scott) = \( \lambda x.\lambda y.x \)

• Everything is encoded as a function
  ▪ So we can easily misuse combinators
    - false 0 if 0 then ... etc...
  ▪ This is no better than assembly language!

What is a Type System?

• A type system is some mechanism for distinguishing good programs from bad
  ▪ Good programs = well typed
  ▪ Bad programs = ill typed or not typable

• Examples:
  ▪ 0 + 1 // well typed
  ▪ false 0 // ill-typed: can’t apply a boolean
  ▪ 1 + (if true then 0 else false) // ill-typed: can’t add boolean to integer

A Definition of Type Systems

“A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.”

– Benjamin Pierce, Types and Programming Languages
### Simply-Typed Lambda Calculus

- \( e ::= n \mid x \mid \lambda x : t . e \mid e \ e \)
  - Functions include the type of their argument
  - We don’t really need this, but it will come in handy

- \( t ::= \text{int} \mid t \to t \)
  - \( t_1 \to t_2 \) is the type of a function that, given an argument of type \( t_1 \), returns a result of type \( t_2 \)
    - \( t_1 \) is the domain, and \( t_2 \) is the range

### Type Judgments

- Our type system will prove judgments of the form
  - \( A \vdash e : t \)
  - “In type environment \( A \), expression \( e \) has type \( t \)”

### Type Environments

- A **type environment** is a map from variables to types (a kind of symbol table)
  - \( \emptyset \) is the empty type environment
    - A closed term \( e \) is well-typed if \( \emptyset \vdash e : t \) for some \( t \)
    - We’ll abbreviate this as \( \vdash e : t \)
  - \( A, x : t \) is just like \( A \), except \( x \) now has type \( t \)
    - The type of \( x \) in \( A, x : t \) is \( t \)
    - The type of \( z \neq x \) in \( A, x : t \) in the type of \( z \) in \( A \)
  - When we see a variable in a program, we look in the type environment to find its type

### Type Rules

\[
\begin{align*}
A \vdash n : \text{int} & \quad & x \in \text{dom}(A) \quad & A \vdash x : A(x) \\
A, x : t \vdash e : t' & & A, x : t \to t' \quad & A \vdash e_1 : t \quad A \vdash e_2 : t \\
& & A \vdash \lambda x : t . e : t \to t' & A \vdash e_1 \ e_2 : t'
\end{align*}
\]
Example

\[ A = - : \text{int} \rightarrow \text{int} \]

\[- \notin \text{dom}(A) \]

\[ A \vdash - : \text{int} \rightarrow \text{int} \quad A \vdash 3 : \text{int} \]

\[ A \vdash 3 : \text{int} \]

Another Example

\[ A = + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \]

\[ B = A, x : \text{int} \]

\[ + \notin \text{dom}(B) \quad x \notin \text{dom}(B) \]

\[ B \vdash + : B \vdash x : i \]

\[ B \vdash + x : \text{int} \rightarrow \text{int} \quad B \vdash 3 : \text{int} \]

\[ A \vdash (\lambda x : \text{int}.+ x) 4 : \text{int} \]

\[ A \vdash (\lambda x : \text{int}.+ x) 4 : \text{int} \]

We’d usually use infix \( x + 3 \)

An Algorithm for Type Checking

- Our type rules are deterministic
  - For each syntactic form, only one possible rule
- They define a natural type checking algorithm
  - \( \text{TypeCheck} : \text{type env} \times \text{expression} \rightarrow \text{type} \)
    - \( \text{TypeCheck}(A, n) = \text{int} \)
    - \( \text{TypeCheck}(A, x) = \text{if } x \in \text{dom}(A) \text{ then } A(x) \text{ else fail} \)
    - \( \text{TypeCheck}(A, \lambda x : t. e) = \text{TypeCheck}((A, x : t), e) \)
    - \( \text{TypeCheck}(A, e_1 e_2) = \)
      - let \( t_1 = \text{TypeCheck}(A, e_1) \) in
      - let \( t_2 = \text{TypeCheck}(A, e_2) \) in
      - if \( \text{dom}(t_1) = t_2 \) then \( \text{range}(t_1) \) else fail

Semantics

- Here is a small-step, call-by-value semantics
  - If an expression can’t be evaluated any more and is not a value, then it is stuck
  
  \[
  (\lambda x.e) v_2 \rightarrow e_1[v_2|x]
  \quad e_1 \rightarrow e_1’
  \quad \]
  
  \[
  e_2 \rightarrow e_2’
  \quad v_1 e_2 \rightarrow v_1 e_2’
  \quad e ::= v \mid x \mid e e
  \quad v ::= n \mid \lambda x : t.e \quad \text{values – not evaluated}
  \]
Progress

- Suppose \( \vdash e : t \). Then either \( e \) is a value, or there exists \( e' \) such that \( e \rightarrow e' \)
- Proof by induction on \( e \)
  - Base cases \( n, \lambda x.e \) – these are values, so we’re done
  - Base case \( x \) – can’t happen (empty type environment)
  - Inductive case \( el \ e2 \) – If \( el \) is not a value, then by induction we can evaluate it, so we’re done, and similarly for \( e2 \). Otherwise both \( el \) and \( e2 \) are values. Inspection of the type rules shows that \( el \) must have a function type, and therefore must be a lambda since it’s a value. Therefore we can make progress.

Preservation

- If \( \vdash e : t \) and \( e \rightarrow e' \) then \( \vdash e' : t \)
- Proof by induction on \( e \rightarrow e' \)
  - Induction (easier than the base case!). Expression \( e \) must have the form \( el \ e2 \).
  - Assume \( \vdash el \ e2 : t \) and \( el \ e2 \rightarrow e' \). Then we have \( \vdash el : t' \rightarrow t \) and \( \vdash e2 : t' \).
  - Then there are three cases.
    - If \( el \rightarrow el' \), then by induction \( \vdash el : t' \rightarrow t \), so \( el' \ e2 \) has type \( t \)
    - If reduction inside \( e2 \), similar

Preservation, cont’d

- Otherwise \( (\lambda x.e) \ v \rightarrow e[v/x] \). Then we have

\[
\frac{x: t' \vdash e : t}{\vdash \lambda x.e : t' \rightarrow t}
\]

- Thus we have
  - \( x : t' \vdash e : t \)
  - \( \vdash v : t' \)
- Then by the substitution lemma (not shown) we have
  - \( \vdash e[v/x] : t \)
- And so we have preservation

Substitution Lemma

- If \( A \vdash v : t \) and \( A, x : t \vdash e : t' \), then \( A \vdash e[v/x] : t' \)
- Proof: Induction on the structure of \( e \)
- For lazy semantics, we’d prove
  - If \( A \vdash el : t \) and \( A, x : t \vdash e : t' \), then \( A \vdash e[el/x] : t' \)
Soundness

- So we have
  - Progress: Suppose $\vdash e : t$. Then either $e$ is a value, or there exists $e'$ such that $e \rightarrow e'$
  - Preservation: If $\vdash e : t$ and $e \rightarrow e'$ then $\vdash e' : t$
- Putting these together, we get soundness
  - If $\vdash e : t$ then either there exists a value $v$ such that $e \rightarrow^* v$, or $e$ diverges (doesn’t terminate).
- What does this mean?
  - Evaluation getting stuck is bad, so
  - “Well-typed programs don’t go wrong”

Product Types (Tuples)

- Self application is not checkable in our system
  - It would require a type $t$ such that $t = t \rightarrow t'$
- The simply-typed lambda calculus is strongly normalizing
  - Every program has a normal form
  - I.e., every program halts!
Recursive Types

- We can type self application if we have a type to represent the solution to equations like $t = t \rightarrow t'$
  - We define the type $\mu\alpha.t$ to be the solution to the (recursive) equation $\alpha = t$
  - Example: $\mu\alpha.\text{int} \rightarrow \alpha$

Folding and Unfolding

- We can check type equivalence with the previous definition
  - Standard unification, omit occurs checks
- Alternative solution:
  - The programmer puts in explicit fold and unfold operations to expand/contract one “level” of the type trees
    - $\text{unfold } \mu\alpha.t = t[\mu\alpha.t/\alpha]$
    - $\text{fold } t[\mu\alpha.t/\alpha] = \mu\alpha.t$

Fold-based Recursive Types

- $e ::= \ldots | \text{fold } e | \text{unfold } e$

ML Datatypes

- Combines fold/unfold-style recursive and sum types
  - Each occurrence of a type constructor when producing a value corresponds to occurrences of $\text{inL/inR}$ and, when recursion is involved, fold
  - Each occurrence of a type constructor in a pattern match corresponds to a case and, when recursion is involved, (at least one) unfold
**ML Datatypes Example**

- **Type list**: `Int of int | Cons of int * int list`
  - Equivalent to `\alpha.\text{int}+(\text{int} \times \alpha)`
- `(Int 3)` equivalent to
  - `\text{fold} (\text{inL}_{\text{int}}\mu.\text{int}+(\text{int} \times \beta) 3)`
- `(Cons (2,(Int 3)))` equivalent to
  - `\text{fold} (\text{inR}_{\text{int}} 2, \text{fold} (\text{inL}_{\text{int}}\mu.\text{int}+(\text{int} \times \beta)))`
- `match e with Int x -> e1 | Cons x -> e2` same as
  - `\text{case} (\text{unfold} e)
    - x:int \rightarrow e1
    - | x:\text{int} \times (\mu.\text{int}+(\text{int} \times \beta)) \rightarrow e2`

**Discussion**

- In the pure lambda calculus, every term is typable with recursive types
  - (Pure = variables, functions, applications only)
- Most languages have some kind of “recursive” type
  - E.g., for data structures like lists, tree, etc.
- However, usually two recursive types that define the same structure but use a different name are considered different
  - E.g., `struct foo { int x; struct foo *next; }` is different from `struct bar { int x; struct bar *next; }`

**Recap**

- We’ve discussed simple types so far
  - Integers, functions, pairs, unions
  - Extensions for recursive types and updatable refs
- Type systems have nice properties
  - Type checking is straightforward (needs annotations)
  - Well typed programs don’t go “wrong”
    - They don’t get stuck in the operational semantics
- But...We can’t type check all good programs

**Up Next: Improving Types**

- How can we build more flexible type systems?
  - More programs type check
  - Type checking is still tractable
- How can reduce the annotation burden?
  - Type inference
Parametric Polymorphism

• Observation: \( \lambda x. x \) returns its argument exactly and places no constraints on the type of \( x \)
  - The identity function works for any argument type

• We can express this with universal quantification:
  - \( \lambda x. x : \forall \alpha. \alpha \rightarrow \alpha \)
  - For any type \( \alpha \), the identity function has type \( \alpha \rightarrow \alpha \)
  - This is also known as parametric polymorphism

System F: annotated polymorphism

• Let’s extend our system as follows:
  - \( t ::= \alpha \mid \text{int} \mid t \rightarrow t \mid \forall \alpha. t \)
  - \( e ::= n \mid x \mid \lambda x.e \mid e \ e \mid \forall \alpha.e \mid e [t] \)

• That is, we add polymorphic types, and we add explicit type abstraction (generalization) …
  - Annotated code locations at which a value of polymorphic type is created
  - … and type application (instantiation)
    - Explicitly annotated code locations at which a value of polymorphic type is used

• This system due to Girard, concurrently Reynolds

Defining Polymorphic Functions

• Polymorphic functions map types to terms
  - Normal functions map terms to terms

• Examples
  - \( \forall \alpha. \lambda x: \alpha. x : \forall \alpha. \alpha \rightarrow \alpha \)
  - \( \forall \alpha. \forall \beta. \lambda x: \alpha. \lambda y: \beta. x : \forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha \)
  - \( \forall \alpha. \forall \beta. \lambda x: \alpha. \lambda y: \beta. y : \forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \beta \)

Instantiation

• When we use a parametric polymorphic type, we apply (or instantiate) it with a particular type
  - In System F this is done by hand:
    - \( (\forall \alpha. \lambda x: \alpha. x)[t1] : t1 \rightarrow t1 \)
    - \( (\forall \alpha. \lambda x: \alpha. x)[t2] : t2 \rightarrow t2 \)

• This is where the term parametric comes from
  - The type \( \forall \alpha. \alpha \rightarrow \alpha \) is a “function” in the domain of types, and it is passed a parameter at instantiation time
• Notice that there are no constructs for manipulating values of polymorphic type
  • This justifies instantiation with any type—that’s what the forall means!
• Note also that we are adding $\alpha$ to $A$; we could (should?) use this to ensure types are well-formed

**Free Variables, Again**

• We’re going to need to perform substitutions on quantified types
  • So just like with lambda calculus, we need to worry about free variables and capture-free substitution
• Define the free variables of a type
  • $FV(\alpha) = \{\alpha\}$
  • $FV(c) = \emptyset$
  • $FV(t \to t') = FV(t) \cup FV(t')$
  • $FV(\forall \alpha. t) = FV(t) - \{\alpha\}$

**Substitution, Again**

• Define $t[u\alpha]$ as
  • $\alpha[u\alpha] = u$
  • $\beta[u\alpha] = \beta$ where $\beta \neq \alpha$
  • $(t \to t')[u\alpha] = t[u\alpha] \to t'[u\alpha]$
  • $(\forall \beta. t)[u\alpha] = \forall \beta. (t[u\alpha])$ where $\beta \neq \alpha$ and $\beta \notin FV(u)$
• Define $e[u\alpha]$ as
  • $(\lambda x: t . e)[u\alpha] = \lambda x: t[u\alpha]. e[u\alpha]$
  • $(\lambda \beta. e)[u\alpha] = \lambda \beta. e[u\alpha]$ where $\beta \neq \alpha$ and $\beta \notin FV(u)$
  • $(e_1 e_2)[u\alpha] = e_1[u\alpha] e_2[u\alpha]$
  • $x[u\alpha] = x$ and $n[u\alpha] = n$
Type Inference

- Let's consider the simply typed lambda calculus with integers
  - $e ::= n \mid x \mid \lambda x:e \mid e \, e$
  - (No parametric polymorphism)

- Type inference: Given a bare term (with no type annotations), can we reconstruct a valid typing for it, or show that it has no valid typing?

Type Language

- Problem: Consider the rule for functions
  $$\begin{align*}
  A, x:t &\vdash e : t' \\
  \hline
  A &\vdash \lambda x.e : t \to t'
  \end{align*}$$

  - Without type annotations, where do we get $t$?
    - We'll use type variables to stand for as-yet-unknown types
      - $t ::= \alpha | \text{int} | t \to t$
    - We'll generate equality constraints $t = t$ among the types and type variables
      - And then we'll solve the constraints to compute a typing

Type Inference Rules

- $\frac{x \in \text{dom}(A)}{A \vdash n : \text{int}}$
- $\frac{A \vdash x : A(x)}{A \vdash \lambda x.e : \alpha \to \beta}$
- $\frac{A \vdash e_1 : t_1 \quad A \vdash e_2 : t_2}{A \vdash e_1 \, e_2 : \beta}$
  - “Generated” constraint

Example

- $\frac{A, x:\alpha \vdash x:\alpha}{A \vdash (\lambda x.x) : \alpha \to \alpha}$
- $\frac{A \vdash 3 : \text{int} \quad \alpha \to \alpha = \text{int} \to \beta}{A \vdash (\lambda x.x) \, 3 : \beta}$

  - We collect all constraints appearing in the derivation into some set $C$ to be solved
  - Here, $C$ contains just $\alpha \to \alpha = \text{int} \to \beta$
    - Solution: $\alpha = \text{int} = \beta$
  - Thus this program is typable, and we can derive a typing by replacing $\alpha$ and $\beta$ by $\text{int}$ in the proof
Solving Equality Constraints

- We can solve the equality constraints using the following rewrite rules, which reduce a larger set of constraints to a smaller set:
  - \( C \cup \{\text{int} = \text{int}\} \Rightarrow C \)
  - \( C \cup \{\alpha = t\} \Rightarrow C[t\alpha] \)
  - \( C \cup \{t = \alpha\} \Rightarrow C[t\alpha] \)
  - \( C \cup \{t_1 \rightarrow t_2 = t_1' \rightarrow t_2'\} \Rightarrow C \cup \{t_1 = t_1'\} \cup \{t_2 = t_2'\} \)
  - \( C \cup \{\text{int} = t_1 \rightarrow t_2\} \Rightarrow \text{unsatisfiable} \)
  - \( C \cup \{t_1 \rightarrow t_2 = \text{int}\} \Rightarrow \text{unsatisfiable} \)

Termination

- We can prove that the constraint solving algorithm terminates.
- For each rewriting rule, either:
  - We reduce the size of the constraint set
  - We reduce the number of “arrow” constructors in the constraint set
- As a result, the constraint always gets “smaller” and eventually becomes empty
  - A similar argument is made for strong normalization in the simply-typed lambda calculus

Occurs Check

- We don’t have recursive types, so we shouldn’t infer them
- So in the operation \( C[t\alpha]\), require that \( \alpha \notin \text{FV}(t) \)
  - (Except if \( t = a \), in which case there’s no recursion in the types, so unification should succeed)
- In practice, it may better to allow \( \alpha \in \text{FV}(t) \) and do the occurs check at the end
  - But that can be awkward to implement

Unifying a Variable and a Type

- Computing \( C[t\alpha] \) by substitution is inefficient
- Instead, use a union-find data structure to represent equal types
  - The terms are in a union-find forest
  - When a variable and a term are equated, we union them so they have the same ECR (equivalence class representative)
    - Want the ECR to be the concrete type with which variables have been unified, if one exists. Can read off solution by reading the ECR of each set.
Example

\[ \alpha \rightarrow \text{int} \quad \beta \rightarrow \text{int} \quad \gamma \rightarrow \text{int} \]

\[ \alpha = \text{int} \rightarrow \beta \quad \gamma = \text{int} \rightarrow \text{int} \quad \alpha = \gamma \]

Unification

- The process of finding a solution to a set of equality constraints is called \textit{unification}
  - Original algorithm due to Robinson
    - But his algorithm was inefficient
  - Often written out in different form
    - See Algorithm W
  - Constraints usually solved on-line
    - As type inference rules applied

Discussion

- The algorithm we’ve given finds the \textit{most general type} of a term
  - Any other valid type is “more specific,” e.g.,
    - \[ \lambda x . x : \text{int} \rightarrow \text{int} \]
  - Formally, any other valid type can be gotten from the most general type by applying a substitution to the type variables
- This is still a \textit{monomorphic} type system
  - \( \alpha \) stands for “some particular type, but it doesn’t matter exactly which type it is”

Inference for Polymorphism

- We would like to have the power of System F, and the ease of use of type inference
  - In short: given an untyped lambda calculus term, can we discover the annotations necessary for typing the term in System F, if such a typing is possible?
  - Unfortunately, no. This problem has been shown to be undecidable.
- Can we at least perform some type inference for parametric polymorphism?
  - Yes. A sweet spot was found by Hindley and Milner
  - But first, let’s consider the general case …
Attempting Type Inference

• Let’s extend simply-typed calculus as follows:
  - \( t ::= \alpha | \text{int} | t \rightarrow t | \forall \alpha . t \)
  - \( e ::= n | x | \lambda x. e | e e \)

• Type inference will automatically infer where to generalize a term, to introduce polymorphic types, and where to instantiate them.

Instantiation

\[
\frac{A \vdash e : \forall \alpha . t}{A \vdash e : t[\alpha \mapsto t']}
\]

• This rule is exactly the same as System F, but we just “magically” pick which \( t' \) to instantiate with.

• You’re surely wondering about algorithmics. We’ll get to that …

Generalization

• Question: When is it safe to generalize (quantify) a type variable \( \alpha \) in the type of expression \( e \)?

• Answer: Whenever we can redo the typing proof for \( e \), choosing \( \alpha \) to be anything we want, and still have a valid typing proof.

Examples

\[
\begin{align*}
A, x : \alpha & \vdash e : \alpha \\
& \vdash \lambda x . x : \alpha \rightarrow \alpha \\
& \vdash \lambda x . x : \text{int} \rightarrow \text{int} \\
& \vdash x : (\text{int} \rightarrow \text{int}) \\
& \vdash \lambda x . x : (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}) \\
\end{align*}
\]

• The choice of the type of \( x \) is purely local to type checking \( \lambda x . x \)
  - There is no interaction with the outside environment
  - Thus we can generalize the type of \( x \)
Examples (cont’d)

- The function restricts the type of \( x \), so we cannot introduce a type variable
  - Thus we cannot generalize the type of \( x \)
  - We can only generalize when the function doesn’t “look at” its parameter

\[
\begin{align*}
A, x : \text{int} & \vdash x : \text{int} \\
A & \vdash \lambda x.x + 3 : \text{int} \to \text{int}
\end{align*}
\]

Generalization Rule

\[
\begin{align*}
A & \vdash e : t \\
\alpha \notin \text{FV}(A) & \\
A & \vdash e : \forall \alpha.t
\end{align*}
\]

- We can generalize any type variable that is unconstrained by the environment
  - Warning: This won’t quite work with refs

Examples (cont’d)

- The choice of the type of \( x \) depends on the type environment
  - In the first derivation, \( x \) and \( y \) have the same type; if we generalize the type of \( x \), they could have different types
  - Thus we cannot generalize the type of \( x \)

\[
\begin{align*}
A, y : \alpha, x : \alpha & \vdash \text{if } p \text{ then } x \text{ else } y : \alpha \\
A, y : \alpha & \vdash \lambda x.\text{if } p \text{ then } x \text{ else } y : \alpha \to \alpha
\end{align*}
\]

Another Justification

- Suppose we have
  - \( A \vdash e : t \) and \( \alpha \notin \text{FV}(A) \)

- Then let \( u \) be any type. By induction, can show
  - \( A[u \backslash \alpha] \vdash e : t[u \backslash \alpha] \)
  - But then since \( \alpha \notin \text{FV}(A) \), that’s equivalent to
  - \( A \vdash e : t[u \backslash \alpha] \)
**Polymorphic Type Inference**

- We’d like to extend our algorithm to polymorphic type inference
  - Performance generalization and instantiation automatically (and deterministically)

- Major problem: Our system for polymorphism is too expressive

---

**Hindley-Milner Polymorphism**

- Restrict polymorphism to only the “top level”
  - Introduce polymorphism at `let`
  - Fully instantiate at use of a polymorphic type

- Here is our new language
  - \( e ::= n \mid x \mid \lambda x.e \mid e1.e2 \mid \text{let } x = e \text{ in } e \)  
  - \( t ::= \alpha \mid \text{int} \mid t \rightarrow t \)  
  - \( s ::= t \mid \forall \alpha.s \)
    - These are type schemes

  - Notice that, according to the prior instantiation rule, we won’t instantiate \( \alpha \) with a scheme \( s \), only a type \( t \)

---

**Old Type Inference Rules**

- \( A \vdash n : \text{int} \)
- \( A, x: \alpha \vdash e \vdash t' \quad \alpha \text{ fresh} \)
  - \( A \vdash \lambda x.e : \alpha \rightarrow t' \)
- \( A \vdash e1 : t1 \quad A \vdash e2 : t2 \)
  - \( t1 = t2 \rightarrow \beta \quad \beta \text{ fresh} \)
  - \( A \vdash e1 \; e2 : \beta \)

---

**New Type Inference Rules**

- At `let`, generalize over all possible variables
  - \( A \vdash e1 : t1 \quad A, x: \forall \alpha.t1 \vdash e2 : t2 \quad \bar{\alpha} = \text{FV}(t1) - \text{FV}(A) \)
  - \( A \vdash \text{let } x = e1 \text{ in } e2 : t2 \)

- At variable uses, instantiate to all fresh types
  - \( A(x) = \forall \bar{\alpha}.t \quad \bar{\beta} \text{ fresh} \)
  - \( A \vdash x : t[\bar{\beta} \backslash \bar{\alpha}] \)

- Here the \( \bar{\alpha} \) denotes a list of type variables
Algorithm W

- A type inference algorithm that explicitly solves the equality constraints on-line
- Instead of implicit global substitution (like we used before), threads the substitution through the inference
- In practice, use previous algorithm, plus generalize at let and instantiate at variable uses.
  - Solve for the type of \( e_1 \), generalize it, then instantiate its solution when doing inference on \( e_2 \)

Example

- Parametric polymorphic type inference
  
  \[
  \text{let } x = \lambda x.x \text{ in } \quad // \quad x : \forall \alpha. \alpha \rightarrow \alpha
  \]
  
  \[
  x \; 3; \quad // \quad x : \beta \rightarrow \beta, \quad \beta = \text{int}
  \]
  
  \[
  x \; (\lambda y.y) \quad // \quad x : \gamma \rightarrow \gamma, \quad \gamma = \delta \rightarrow \delta
  \]

- This would be untypable in a monomorphic type system

Kinds of Polymorphism

- We’ve just seen parametric polymorphism
  - System F and Hindley-Milner style polymorphism
- Another popular form is subtype polymorphism
  - As in OO programming
  - These two can be combined (e.g., Java Generics)
- Some languages also have \textit{ad-hoc polymorphism}
  - E.g., + operator that works on ints and floats
  - E.g., overloading in Java

An Imperative Language

\[
e ::= x | \lambda x.e | e \; e
\]

\[
| \quad \text{ref } e \quad \text{allocation}
| \quad !e \quad \text{dereference}
| \quad e := e \quad \text{assignment}
| \quad e; e \quad \text{sequencing}
\]

- Notice that this is not C
  - Variables cannot be updated; only references can
  - I.e., there are no l-values or r-values
- This is a language with \textit{updatable references}
Examples

!(ref 0)

let x = ref 0 in
  x := !x + 1

let x = ref 0 in
  \( y. x := !x + 1; !x \)

Type Checking Rules

- \( t ::= \ldots \mid \text{ref } t \)
  - Note: in ML this type is written \( t \text{ ref} \)

\[
\frac{A \vdash e : t}{A \vdash \text{ref } e : \text{ref } t} \quad \frac{A \vdash e : \text{ref } t}{A \vdash !e : t}
\]

\[
\frac{A \vdash e_1 : \text{ref } t \quad A \vdash e_2 : t}{A \vdash e_1 := e_2 : t}
\]

Unit and the Unit Type

- Sometimes in imperative programs we write expressions that have some side effect but no interesting result
  - To represent this directly, use \textit{unit}:
    - \( e ::= \ldots \mid () \)
    - \( t ::= \ldots \mid \text{unit} \)

\[
\frac{}{A \vdash () : \text{unit}} \quad \frac{A \vdash e_1 : \text{ref } t \quad A \vdash e_2 : t}{A \vdash e_1 := e_2 : \text{unit}}
\]

Operational Semantics

- Now we need to keep track of memory
  - State is a map from locations to values
  - Our redexes will be tuples \( \langle \text{State}, \text{expression} \rangle \)
  - As a consequence, order of evaluation matters

- As before, evaluation will yield a fully-evaluated term, also called a \textit{value}
  - \( v ::= x \mid \lambda x. e \)
  - \( e ::= v \mid e \mid \text{ref } e \mid !e \mid e := e \)
Operational Semantics (cont’d)

\[ \langle S, (\lambda x. e) \rangle \rightarrow \langle S', (\lambda x. e) \rangle \]

\[ \langle S, e \rangle \rightarrow \langle S', v \rangle \quad \text{loc fresh} \]
\[ \langle S, \text{ref } e \rangle \rightarrow \langle S[v\backslash \text{loc}], \text{loc} \rangle \]

Polymorphism and References

- Suppose we want polymorphism in our imperative language
  - \( e ::= x | n | \lambda x.e | e e | \text{ref } e | !e | e := e \)
  - \( s ::= t | \forall \alpha.s \)
  - \( t ::= \alpha | \text{int} | t \rightarrow t | \text{ref } t \)

- What if we try our standard rule?
  - \( A \vdash e_1 : t_1 \quad A, x : \forall \tilde{\alpha}. t_1 \vdash e_2 : t_2 \quad \tilde{\alpha} = \text{FV}(t_1)\backslash \text{FV}(A) \)
  - \( A \vdash \text{let } x = e_1 \text{ in } e_2 : t_2 \)

- Example (due to Tofte)
  - \( \text{let } r = \text{ref } (\lambda x.x) \text{ in } \quad r : \forall \alpha. \text{ref } (\alpha \rightarrow \alpha) \)
    - \( r := \lambda x.x + 1; \quad \text{checks; use } r \text{ at ref } (\text{int} \rightarrow \text{int}) \)
    - \( (!r) \text{ true } \quad \text{oops! checks; use } r \text{ at ref } (\text{bool} \rightarrow \text{bool}) \)

- \( \alpha \) should not be generalized, because later uses of \( r \) may place constraints on it

- Nobody realized there was a problem for a long time

Naive Generalization is Unsound
**Solution: The Value Restriction**

- Only allow values to be generalized
  - $v ::= x \mid n \mid \lambda x.e$
  - $e ::= v \mid e \; e \mid \text{ref} \; e \mid !e \mid e := e$

\[
A \vdash v : t_1 \quad A, x : \forall \alpha. t \vdash e_2 : t_2 \quad \alpha = \text{FV}(t) - \text{FV}(A)
\]

\[
A \vdash \text{let} \; x = v \; \text{in} \; e_2 : t_2
\]

- Intuition: Values cannot later be updated
- This solution due to Wright and Felleisen
  - Tofte found a much more complicated solution

**Benefits of Type Inference**

- Handles higher-order functions
- Handles data structures smoothly
- Works in infinite domains
  - Set of types is unlimited
- No forward/backward distinction
- Polymorphism provides context-sensitivity

**Drawbacks to Type Inference**

- Flow-insensitive
  - Types are the same at all program points
  - May produce coarse results
  - Type inference failure can be hard to understand

- Polymorphic type inference may not scale
  - Exponential in worst case
  - Seems fine in practice (witness ML)

- Flow-insensitive
  - Types are the same at all program points
  - May produce coarse results
  - Type inference failure can be hard to understand

- Polymorphism may not scale
  - Exponential in worst case
  - Seems fine in practice (witness ML)